Searching Long Repeats in Streams

Oleg Merkurev
Ural Federal University, Ekaterinburg, Russia
o.merkuryev@gmail.com

Arseny M. Shur
Ural Federal University, Ekaterinburg, Russia
arseny.shur@urfu.ru

Abstract

We consider two well-known related problems: Longest Repeated Substring (LRS) and Longest Repeated Reversed Substring (LRRS). Their streaming versions cannot be solved exactly; we show that only approximate solutions by Monte Carlo algorithms are possible, and prove a lower bound on consumed memory. For both problems, we present purely linear-time Monte Carlo algorithms working in $O(E + \frac{n}{E})$ space, where E is the additive approximation error. Within the same space bounds, we then present nearly real-time solutions, which require $O(\log n)$ time per symbol and $O(n + \frac{n}{E} \log n)$ time overall. The working space exactly matches the lower bound whenever $E = O(n^{0.5})$ and the size of the alphabet is $\Omega(n^{0.01})$.

2012 ACM Subject Classification Theory of computation → Streaming, sublinear and near linear time algorithms

Keywords and phrases Longest repeated substring, longest repeated reversed substring, streaming algorithm, Karp–Rabin fingerprint, suffix tree

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.31

Funding *Oleg Merkurev*: Supported by the Russian Ministry of Education and Science, project 1.3253.2017.
Arseny M. Shur: Supported by the Russian Ministry of Education and Science, project 1.3253.2017.

Acknowledgements The authors are grateful to D. Kosolobov for very useful comments.

1 Introduction

The streaming model of computation became popular in string processing during the last decade. In this model, there is no random access to the input: the input string of length n arrives symbol by symbol and the space available to the algorithm is sublinear in n. In the design of algorithms for this model, the two main goals are minimization of the total space required and of the worst-case time in which a single symbol is processed (update time). If a problem can be solved only approximately, there is usually a trade-off between the approximation error and the required space/time. A wide use of Monte Carlo randomized algorithms is another distinctive feature of the streaming model.

Surprisingly, the exact pattern matching, which is the fundamental problem in stringology, can be solved in the streaming model very efficiently. The first efficient algorithm was presented by Porat and Porat [18]; soon after that Breslauer and Galil showed [3] that $O(\log m)$ space is enough to find all occurrences of a length-m pattern in a length-n text in real time (=constant update time), just with a small probability of false positive. More complicated pattern matching problems, like approximate or multiple matching, were also analysed [5, 6, 7, 12, 13, 19], as well as the related problem of estimating the Hamming distance between the pattern and length-m substrings of the text [8].

Another class of string problems concerns the search of repetitions (periods, repeats, palindromes, etc). As shown in [11], a longest palindrome in a stream can be found approximately by a real-time Monte Carlo algorithm spending $O(M)$ words of memory,

where $M = \frac{n}{E}$ for the additive error E and $M = \frac{\log n}{\log(1+\varepsilon)}$ for the multiplicative error $1+\varepsilon$; matching lower bounds for the required space were also proved there. A longest palindrome with at most d errors can be found within the space and update time which are both polylog in n and inverse polynomial in the error [14].

In this paper, we consider the problem of finding the longest string occurring at least twice in the input (longest repeating substring, LRS) and the problem of finding the longest string occurring in the input together with its reversal (longest repeating reversed substring, LRRS). Solving LRS in the standard RAM model in linear time is a well-known application of the suffix tree, first mentioned in the pioneering paper by Weiner [21]. LRRS can be solved in the same way (e.g., by building the suffix tree for the reversal of the input string). Note that LRS and LRRS also admit efficient parallel algorithms [2]. However, in the streaming model these problems can be solved only approximately and only with high probability, as shown in Section 6. Our main contributions are Monte Carlo algorithms for both problems, solving them with an additive approximation error E in $O(\frac{n}{E} + E)$ space. We give two algorithms for each problem; "simple" algorithms work in $O(n)$ total time, but have the update time $O(\frac{n}{E})$, while more elaborate versions use $O(n + \frac{n}{E}\log n)$ total time and allow updates in $\mathcal{O}(\log n)$ time. After preliminaries, we describe the main idea of reaching sublinear memory in Section 3, algorithms for LRRS in Section 4, algorithms for LRS in Section 5, and memory lower bounds in Section 6.

2 Model and Definitions

Let S denote a string of length n over an alphabet $\Sigma = \{1, \ldots, N\}$, where N is polynomial in n. We write $S[i]$ for the ith symbol of S and $S[i..j]$ for its *substring* (or *factor*) $S[i]S[i+1]\cdots S[j]$; thus, $S[1..n] = S$. A *prefix* (resp. *suffix*) of S is a substring of the form $S[1..j]$ (resp., $S[j..n]$). A *period* of S is a positive integer p such that $S[1..n-p] = S[p+1..n]$; here the string $S[1..n-p]$ is a *border* of S. The *reversal* of S is the string $\bar{S} = S[n]S[n-1]\cdots S[1]$. If $S = \bar{S}$, then S is a *palindrome*. A repeat in S is a pair of equal substrings $S[i..i+l-1] = S[j..j+l-1]$, where $i < j$; we denote repeats by the triples (i, j, l) and write $L(S)$ for the maximum length of a repeat in S. Similar, the condition $S[i..i+l-1] = \bar{S}[j..j+l-1]$, $i \le j$, defines the *reversed repeat* (i, j, l) in S; we write $\bar{L}(S)$ for the maximum length of a reversed repeat in S. Note that a palindrome $S[i..i+l-1]$ is the reversed repeat (i, i, l); moreover, if $i + l \ge j - 1$ for a reversed repeat (i, j, l), then $S[i..j+l-1]$ is a palindrome of length $l + j - i$. Thus the longest reversed repeat in S is either non-overlapping (the "left occurrence" $S[i..i+l-1]$ ends before the "right occurrence" $S[j..j+l-1]$ begins), or a palindrome.

We write log for the binary logarithm and $a \overline{\bmod} b$ instead of $(a-1) \bmod b + 1$.

We work in the *streaming model* of computation: the input string $S[1..n]$ (the *stream*) is read left to right, one symbol at a time, and cannot be stored, because the available space is sublinear in n. The space is counted as the number of $\mathcal{O}(\log n)$-bit machine words. The *update* time is the worst-case time spent between two reads.

An approximation algorithm for a maximization problem has *additive error* E if it finds a solution with the cost at least $OPT - E$, where OPT is the cost of optimal solution; here E can be a function of the size of the input. For an instance LRS(S) of the LRS problem, $OPT = L(S)$; similar for LRRS(S) and $\bar{L}(S)$.

A *Las Vegas algorithm* always returns a correct answer, but its working time and memory usage on length-n inputs are random variables. A *Monte Carlo algorithm* gives a correct answer with high probability (at least $1 - \frac{1}{n}$) and has deterministic working time and space.

▶ **Observation 1.** *A longest palindrome in a stream can be approximated in $\mathcal{O}(\frac{n}{E})$ space and $\mathcal{O}(1)$ update time by Monte Carlo Algorithm A of [11], which is better than we can achieve for* LRRS*. So for an algorithm designed to solve* LRRS *it suffices to process only* some *reversed repeats (including* all non-overlapping ones*) in a stream; we assume that the algorithm for the longest palindrome is run in parallel, and the longer of two results is returned.*

Karp–Rabin fingerprints [15] is a hash function widely used for streaming string algorithms. Let p be a fixed prime from the range $[n^{3+\alpha}, n^{4+\alpha}]$ for some $\alpha > 0$, and r be a fixed integer randomly chosen from $\{1, \dots, p-1\}$. For a string S, its forward hash and reversed hash are defined, respectively, as $\phi^F(S) = \left(\sum_{i=1}^{n} S[i] \cdot r^i\right) \bmod p$ and $\phi^R(S) = \left(\sum_{i=1}^{n} S[i] \cdot r^{n-i+1}\right) \bmod p$. Clearly, the forward hash of a string coincides with the reversed hash of its reversal; this fact is used to detect reversed repeats. The probability of hash collision for two strings of length m is at most m/p; thus, a linear-time algorithm faces a collision with probability $O(n^{-1-\alpha})$ by the choice of p. All further considerations assume that no collisions happen. For an input stream S, we denote $F^F(i,j) = \phi^F(S[i..j])$ and $F^R(i,j) = \phi^R(S[i..j])$. Hashes of substrings can be extracted in constant time from the hashes of prefixes, as the next observation shows.

▶ **Proposition 2** ([3])**.** *The following equalities hold:*

$$F^F(i,j) = r^{-(i-1)} \left(F^F(1,j) - F^F(1,i-1)\right) \bmod p\,,$$
$$F^R(i,j) = F^R(1,j) - r^{j-i+1} F^R(1,i-1) \bmod p\,.$$

For an input stream S, the tuple $I(i) = (i, F^F(1,i-1), F^R(1,i-1), r^{1-i} \bmod p, r^i \bmod p)$ is its *i-th frame*. The proposition below is immediate from definitions and Proposition 2.

▶ **Proposition 3.** *Given $I(i)$ and $S[i]$, the tuple $I(i+1)$ can be computed in $\mathcal{O}(1)$ time.*

Let $\mathcal{T}(S)$ denote the suffix tree of a length-n string S. Recall that $\mathcal{T}(S)$ is the compressed trie of all suffixes of S, has $\mathcal{O}(n)$ nodes and edges, and occupies $\mathcal{O}(n)$ words of memory; S is a part of the data structure. Edges are labelled by substrings of S, each node is identified with the label of the path from the root to it. Not all substrings of S correspond to nodes; in general, substrings are addressed by *positions*. A position in the suffix tree is a pair $pos = (v, raise)$, where v is a node and $raise \geq 0$. This position corresponds to the prefix of v of length $|v| - raise$ and points to the "locus" in $\mathcal{T}(S)$ on the incoming edge of v, $raise$ symbols above v. The walks on the tree are navigated by (direct) *suffix links*; for a node v, $link(v)$ is the longest proper suffix of v. A similar link from an arbitrary position is called *implicit suffix link* and is not stored; computing such links is a crucial primitive for the work with suffix trees.

3 Reduction to Packed Repeats

The main difficulty of LRS and LRRS is that they are not local: the two occurrences of a repeat can be separated by as much as $\Omega(n)$ symbols. To avoid storing the whole input, we hash blocks of some fixed size b, consider hashes as new symbols, called hashletters, and search for repeated strings of hashletters. We define *direct trace* and *reversed trace* of a stream S, with the block size b and shift r, as the strings

$$P_{r,b} = F^F(r, r+b-1)F^F(r+b, r+2b-1) \cdots F^F(r+x_{r,b}b, r+(x_{r,b}+1)b-1) \text{ and}$$
$$Q_{r,b} = F^R(r+x_{r,b}b, r+(x_{r,b}+1)b-1) \cdots F^R(r+b, r+2b-1)F^R(r, r+b-1),$$

respectively, where $x_{r,b} = \left\lfloor \frac{|S|+1-r}{b} \right\rfloor - 1$ and the alphabet is $0, \dots, p-1$.

Let us fix b and consider only the traces with block size b and shifts $r = 1, \dots, b$, writing P_r, Q_r, x_r instead of $P_{r,b}, Q_{r,b}, x_{r,b}$; the traces P_b, Q_b are called *main*. We say that a substring $S[l..t]$ is contained in the traces P_r and Q_r if $l \equiv_b r$ and $t - l + 1 \equiv_b 0$. This condition means that $S[l..t]$ can be partitioned into blocks such that the corresponding hashletters after the forward (resp., backward) hashing constitute a substring of the trace P_r (resp., Q_r).

We store only the main trace (P_b for LRS and Q_b for LRRS) and search for repeats with the left occurrence contained in the main trace. We call these repeats *packable*. By the length argument, the right occurrence of a packable repeat is also contained in some trace. Long packable repeats approximate the solutions for LRS and LRRS:

▶ **Observation 4.** *For each repeat (reversed repeat) (i, j, l) such that $l > 2b - 2$, a stream S contains a packable repeat (resp., packable reversed repeat) (i', j', l') such that $l' \geq l - 2b + 2$. Namely, take $i' = \lceil \frac{i}{b} \rceil b$, $j' = j + i' - i$, $l' = \lfloor \frac{i+l-i'}{b} \rfloor b$.*

A *packed repeat* in a string S, denoted by a quadruple (r, i_l, i_r, k), is a pair of equal substrings $P_b[i_l \,..\, i_l{+}k{-}1] = P_r[i_r \,..\, i_r{+}k{-}1]$, where $i_l < i_r$ and $1 \leq r \leq b$ (see Fig. 1). Similar, a *packed reversed repeat* is a pair $Q_b[x_r{+}2{-}i_l \,..\, x_r{+}3{-}i_l{-}k] = P_r[i_r \,..\, i_r{+}k{-}1]$.

▶ **Observation 5.** *If (i, j, l) is a packable repeat (packable reversed repeat), then the tuple $(j \bmod b, i/b, \lfloor j/b \rfloor + 1, l/b)$ is a packed repeat (packed reversed repeat). Conversely, if (r, i_l, i_r, k) is a packed repeat (packed reversed repeat), then $(i_l b, (i_r - 1)b + r, kb)$ is a packable repeat (resp., packable reversed repeat), up to a hash collision.*

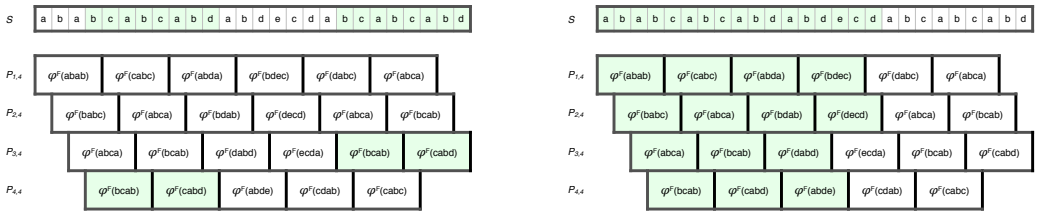

Figure 1 Traces and packed repeats. Left: packable repeat $(4, 19, 8)$ and its packed repeat $(3, 1, 5, 2)$ (colored). Right: hashletters available in all direct traces after reading $S[17]$ (colored).

Due to Observations 4, 5, we reduced the initial problems to the problems of finding a longest packed repeat and a longest packed reversed repeat in a stream; their solutions will solve, w.h.p., LRS and LRRS with an additive error less than $2b$. By Observation 1, the algorithm for packed reversed repeats can skip some of them if the corresponding packable reversed repeats are overlapping.

A hashletter has the form $F^F(j, j + b - 1)$ or $F^R(j, j + b - 1)$. This hashletter becomes *available* when we read the symbols $S[j], S[j{+}1], \dots, S[j{+}b{-}1]$, i.e., after reading $S[j{+}b{-}1]$. Hence, after each read, starting from $S[b]$, a new hashletter becomes available for one trace and one reversed trace. To compute this hashletter $F^F(j, j + b - 1)$ or $F^R(j, j + b - 1)$ it is sufficient to know the frames $I(j + b)$ and $I(j)$. Before reading $S[i]$ we store $I(i - b + 1), I(i - b + 2), \dots, I(i)$. The numbers of these frames are all distinct modulo b, so the frames can be stored in a cyclic queue of length b. The prefix of the trace P_r (resp., the suffix of Q_r) available after reading $S[i]$ is denoted by P_r^i (resp., Q_r^i). Note that $P_r^i = P_r[1..\lfloor \frac{i-r+1}{b} \rfloor]$, similar for Q_r^i.

4 Search of a Longest Reversed Repeat

We first approach LRRS. By Observation 1, it is sufficient to analyse any set of reversed repeats which includes all non-overlapping ones. If a reversed repeat is non-overlapping, then the packable reversed repeat obtained from it (Observation 4) is also non-overlapping.

Let us define the packed counterpart of non-overlapping repeats. A *handy repeat* is a packed reversed repeat (r, i_l, i_r, k) such that its left occurrence is a substring of $Q_b^{i_r b-1}$.

▶ **Lemma 6.** *For a non-overlapping packable reversed repeat, the corresponding packed reversed repeat is handy.*

Proof. Let (i, j, l) be a non-overlapping packable reversed repeat; the corresponding packed reversed repeat is $(j \bmod b, i/b, \lfloor j/b \rfloor{+}1, l/b)$ by Observation 5. Its left occurrence is contained in Q_b^{i+l-1}, and $i + l - 1 < j - 1 < (\lfloor j/b \rfloor + 1)b - 1$, so it is handy by definition. ◀

By Lemma 6, to get the desired approximation to LRRS it is enough to find the longest handy repeat in S. We use Weiner's algorithm [21] to maintain a dynamic suffix tree for Q_b which equals $\mathcal{T}(Q_b^j)$ after processing $S[j]$, for each $j \le n$. Processing a symbol $S[i]$, we add a new hashletter to the trace P_r, where $r = (i{+}1) \bmod b$, and find the longest handy repeat, the right occurrence of which is a suffix of P_r^i; we denote this suffix by $suff_r^i$. Since the repeat is handy, the last hashletter of the left occurrence is added to the tree before the first hashletter of the right occurrence becomes available. This condition implies that if the repeat is longer than one hashletter, it extends a handy repeat found on the iteration $i - b$.

▶ **Observation 7.** *By definition of a handy repeat, the condition $P_r^i = P_r^{i-1}$ implies $suff_r^i = suff_r^{i-1}$. Hence $suff_r^{i-1} = suff_r^{i-b}$ whenever $i + 1 \equiv_b r$. That is, the update of the main trace Q_b does not affect the longest handy repeat ending with a suffix of P_r^i.*

For each $r = 1, \dots, b$, we maintain the position of the string $suff_r^i$ in the tree $\mathcal{T}(Q_b^i)$. This position is denoted by pos_r^i.

▶ **Observation 8.** *The equality $suff_r^{i-1} = suff_r^{i-b}$ stated in Observation 7 does not necessarily imply $pos_r^{i-1} = pos_r^{i-b}$. Namely, these pairs are equal iff it is still valid to refer to the position of $w = suff_r^{i-b}$ in the tree $\mathcal{T}(Q_b^{i-1})$ with the pair $pos_r^{i-b} = (v, raise)$. However new nodes can appear during the last update of the tree, splitting the incoming edge of v such that the position of w will be above some ancestor of v. Still, if we go up by $raise$ symbols from the node v, we will reach the position of w.*

After reading $S[i]$, we need to compute pos_r^i, where $r = (i{+}1) \bmod b$. We may suppose that we have computed pos_r^{i-b} after reading $S[i - b]$. By Observation 8, one can get pos_r^{i-1} from $pos_r^{i-b} = (v, raise)$ walking up the tree from v to the lower end of the edge containing the position which is $raise$ symbols above v.

▶ **Proposition 9.** *If $i + 1 \equiv_b r$, the longest proper prefix of $suff_r^i$ is a suffix of $suff_r^{i-1}$.*

Proof. Let $(r, i_l, \frac{i-r+1}{b} - k + 1, k)$, $k \ge 1$, be the handy repeat corresponding to $suff_r^i$. The longest proper prefix of $suff_r^i$ corresponds to the handy repeat $(r, i_l + 1, \frac{i-r+1}{b} - k + 1, k - 1)$; its right occurrence ends at the position $\frac{i-r+1}{b} - 1$ and thus is a suffix of P_r^{i-b}. Then this occurrence is a suffix of $suff_r^{i-b}$ by definition. By Observation 7, $suff_r^{i-b} = suff_r^{i-1}$. ◀

By Proposition 9, we can find the position pos_r^i in the tree $\mathcal{T}(Q_b^i)$ as follows: take the longest suffix w of $suff_j^{i-1}$ such that its position in the tree has a transition by the hashletter $a = F^F[i{-}b{+}1..i]$, and follow this transition to get pos_r^i; if the root of $\mathcal{T}(Q_b^i)$ has no transition by a, pos_r^i coincides with the root. To find w, we scan suffixes of $suff_j^{i-1}$ in the order of

decreasing length, following suffix links. If the current position is on the edge (u, v), t symbols from its upper end u, then its implicit link can be found by descending t symbols from the node $link(u)$. Finally, if $i \overline{\bmod} b = b - 1$, a new hashletter is added to Q_b, and an iteration of Weiner's algorithm is run to update the suffix tree. Now we can formulate

▶ **Theorem 10.** *There is an algorithm solving, w.h.p.,* LRRS *in a stream with a given additive error $E = \mathcal{O}(n^{0.99})$ in $\mathcal{O}(\frac{n}{E} + E)$ space, $\mathcal{O}(n)$ total time, and $\mathcal{O}(\frac{n}{E})$ update time.*

Proof. Let us fix $b = \lfloor \frac{E}{2} \rfloor$ and write $\mathcal{T}_j$ for the suffix tree $\mathcal{T}(Q_b^j)$. Here we present Algorithm 1 which finds, w.h.p., the longest handy repeat with the block size b in a length-n stream S within the space and time bounds stated in the theorem. Comparing the corresponding packable repeat with the longest palindrome in S found by [11, Algorithm A] and taking the longer of two, we obtain, w.h.p., the solution for LRRS(S) with the error, space, and time bounds as in the theorem; see Observations 1, 4, 5, and Lemma 6.

During the algorithm we maintain a suffix tree $\mathcal{T}$, an array $pos[1..b]$ of positions in $\mathcal{T}$, an array $SI[1..b]$ of recent frames and the last frame I. By *ith iteration* we mean all operations starting with the read of $S[i]$ and preceding the read of $S[i+1]$. After $(i-1)$th iteration, $\mathcal{T} = \mathcal{T}_{i-1}$, $pos[r]$ contains the latest computed value of pos_r^j (i.e., $j \geq i - b$ for each r), and SI contains the frames $I(j)$ for $j = i - b + 1, \ldots, i$ such that $I(j)$ is stored in $SI[(j-1) \overline{\bmod} b]$. The ith iteration looks as follows:

Algorithm 1 : Algorithm AdditiveReversedRepeat, i-th iteration ($i \geq b$).

1: $r = i \bmod b + 1$
2: read $S[i]$; compute $I(i+1)$ from $I(i)$; $I = I(i+1)$
3: compute $FF = F^F(i - b + 1, i)$ from I and $SI[r]$
4: update $pos[r] = pos_r^{i-b}$ to pos_r^{i-1} by walking up
5: compute $pos[r] = pos_r^i$ by traversing $\mathcal{T}$ from pos_r^{i-1} by the hashletter FF
6: update *answer* by the value determined by $pos[r]$
7: **if** $r = b$ **then**
8: compute $FR = F^R(i - b + 1, i)$ from I and $SI[r]$
9: update $\mathcal{T}$ to $\mathcal{T}_i$, appending FR by Weiner's algorithm
10: $SI[r] = I$

The update of $pos[r]$ in lines 4-5 is correct by Observation 8 and Proposition 9. In order to update the answer in line 6, we store in each node of the suffix tree its string depth sd (the length of the corresponding string). Then the new value $(v, raise)$ of $pos[r]$ gives us the length of the corresponding handy repeat as $sd(v) - raise$. From the length and the number of the iteration we get the right occurrence of the repeat. To find the left occurrence, recall that the edge of a suffix tree with the lower end v is labeled by the last position of an occurrence of v in the string; subtracting $sd(v) - 1$ from this position, we get the first position of the left occurrence of the repeat. Thus Algorithm 1 correctly computes the longest handy repeat. Consider its time/space costs. The suffix tree is of size $\mathcal{O}(\frac{n}{E})$, both the array of positions and the array of frames are of size $\mathcal{O}(E)$, giving the required space bound.

We store transitions in the suffix tree in a single hash table with the pair (node, symbol) as the key, using the dynamic perfect hashing [10]. This method provides a dictionary with constant-time lookup; with probability at least $1 - \frac{1}{n}$ all updates work in constant time as well. If an update takes more than a prescribed constant time, we output a "repeat" $(1, 1, n)$ and stop. This results in an improbable error on the same side as the error caused by a hash collision. Thus the descents in the suffix tree require constant time per edge.

At an iteration, potentially heavy computations are in the lines 5 and 9. Each of them, in the worst case, requires the time proportional to the size of the suffix tree, i.e., $O(\frac{n}{E})$. Note that line 4 requires a constant number of operations because Weiner's algorithm adds to the tree at most one internal node per iteration.

To estimate the total time, count the total number of elementary operations in each of the lines 5 and 9 during all iterations. In line 9, a suffix tree is built for a string of length $\mathcal{O}(\frac{n}{E})$ over a polynomial alphabet (p is polynomial in n and thus in $\frac{n}{E}$). This can be done in $\mathcal{O}(\frac{n}{E})$ time using the hash table. Now consider line 5, grouping all iterations with the same trace P_r. Consider the evolution of the string depth of $pos[r]$. Initially, $sd = 0$. Each suffix link transition decreases it by 1; it can increase by 1 once per iteration, if the transition by the current symbol is found. Hence the total number of suffix link transitions is $\mathcal{O}(\frac{n}{E})$, constant time per each. Finally compute the number of descents which follow suffix link transitions. Consider the tree depth td of $pos[r]$. Initially, $td = 0$, and td is bounded by the depth of the tree, which is $\mathcal{O}(\frac{n}{E})$. A suffix link transition decreases td by at most 1; each ascent in line 4 decreases it by 1; updates of the tree do not decrease it. Since the total number of ascents in lines 4 and suffix link transitions in line 5 is $\mathcal{O}(\frac{n}{E})$, the total number of descents is $\mathcal{O}(\frac{n}{E})$ as well. Summing over all traces, we get the total time $\mathcal{O}(n)$, as required.

Thus, LRRS can be solved within the space/time costs stated in the theorem. ◀

Next we modify Algorithm 1 to avoid slow updates.

▶ **Theorem 11.** *There is an algorithm solving, w.h.p.,* LRRS *in a stream with a given additive error $E = \mathcal{O}(n^{0.99})$ in $\mathcal{O}(\frac{n}{E} + E)$ space, $\mathcal{O}(n + \frac{n}{E}\log n)$ total time, and $\mathcal{O}(\log n)$ update time.*

Proof. Algorithm 1 has two heavy parts: updating the suffix tree and searching in it. For the first part, we replace Weiner's construction with its modification by Amir et al. [1]. This modification can work with the polynomial integer alphabet and has $O(\log n)$ update time and $O(n \log n)$ total time, thus adding $\frac{n}{E}\log n$ to the total time cost of our algorithm.

We should note that the algorithm by Amir et al. makes no use of suffix links; instead, the positions where to add new suffixes are determined through queries to the "balanced indexed structure" (BIS). BIS is capable of finding such a position in $\mathcal{O}(\log|\mathcal{T}|)$ time; in addition, it can find the longest common prefix of an arbitrary string with $\mathcal{T}$, also in $\mathcal{O}(\log|\mathcal{T}|)$ time. This allows us to build direct suffix links of new nodes within the same time bounds.

The search will use delayed operations. For each shift $r = 1, \ldots, b$ we maintain a "conveyor" C_r, consisting of the position $pos = pos[r]$ in the tree $\mathcal{T}$, a queue *queue* of hashletters awaiting processing (it can be viewed as a suffix of the string P_r), and a flag $flag$ set to 0 if the algorithm should skip the update of the answer with the current value of *pos*. One iteration of our algorithm is shown below.

A new hashletter is added to the current conveyor in line 3; the function $lazyProcess$, called in line 4, performs two delayed transitions in $\mathcal{T}$ (by suffix links or by queue symbols). If only one transition is available (by the only queue symbol), one transition is performed.

Let $C = C_r$ be the current conveyor, *maxlen* be the maximum string depth among all positions *pos* seen in C (= the length of the longest handy repeat with the right occurrence in P_r, found so far). After each call to $lazyProcess$, the following semi-invariant is kept: $balance = maxlen - sd(pos) - 2|queue| \geq 0$. Adding a symbol to *queue* decreases *balance* by 2; a suffix link transition increases it by 1. A transition by a symbol updates *pos* and so is followed by a deletion from *queue*; thus it increases *balance* by 1 or even by 2 (if the new *pos* changes *maxlen*). Thus if $lazyProcess$ performed two transitions, then *balance* has not decreased, and if there was one transition, then $|queue| = 0$ and $balance \geq 0$ by definition.

Algorithm 2 : FastAdditiveReversedRepeat, ith iteration ($i \geq b$).

$r = i \bmod b + 1$
read $S[i]$; compute $I(i+1)$ from $I(i)$; $I = I(i+1)$
compute $FF = F^F(i-b+1, i)$ from I and $SI[r]$; put (C_r, FF)
$C_r = lazyProcess(C_r)$; $flag = [queue \text{ is empty}]$
if $flag = 1$ **then**
    update *answer* by value determined by C_r
if $r = b$ **then**
    compute $FR = F^R(i-b+1, i)$ from I and $SI[r]$
    update $\mathcal{T}$ to $\mathcal{T}_i$, appending FR and building suffix links using BIS
$SI[r] = I$

Because of the semi-invariant, $|queue| > 0$ implies $sd(pos) + |queue| < maxlen$. This means that after processing the whole queue both *maxlen* and the longest handy repeat cannot be updated. Hence at the end of an iteration $flag$ is set to 0 if $|queue| > 0$ and to 1 otherwise.

▶ **Observation 12.** *Between the iteration when a hashletter was added to a queue, and an iteration when it was processed to get pos, the suffix tree can be updated, and a wrong (too deep) pos can be found. However, while the queue is nonempty, the answer is not updated because $flag = 0$. When it becomes empty, the position pos will be correct, since the newest symbol in the queue appeared after the last update of $\mathcal{T}$.*

Observation 12 implies that Algorithm 2 performs exactly the same updates of the answer as Algorithm 1, and hence is correct. Now consider its time and space costs.

We cannot store *queue* explicitly due to memory restrictions. Instead, we factorize the string *queue* into a prefix (or head) *head* and a suffix (or tail) *tail*, storing them separately. We use four words of memory in total and support three operations in $\mathcal{O}(1)$ time: append a symbol to *tail*; delete the first symbol of *head*; set *head* to *tail* if *head* is empty. This provides full functionality of a queue; let us consider details.

The tail is stored as the position $posT = (v, raise)$ of the string *tail* in the tree $\mathcal{T}$. The head is represented by two numbers L and R such that the interval $[L..R]$ of the main trace Q_b equals *head*; recall that Q_b is a part of the suffix tree data structure. To delete the first hashletter from the queue, we increment L by 1. If now $L > R$ (= the head is empty), we compute the interval of Q_b equal to *tail*: if $posT = (v, raise)$, the incoming edge of v is marked by the last position x of an occurrence of v is Q_b, so $tail = Q_b[x-sd(v)+1..x-raise]$. Then we set $[L..R]$ to this interval and $posT$ to $(root, 0)$, thus moving *tail* to *head*.

To add a hashletter c to the queue, we try to make a transition by c from $posT$. If it succeeds, we just update $posT$ with the position reached. If it fails, the string $tail \cdot c$ does not occur in Q_b and we update the current conveyor as follows: $pos = posT$; $posT = (root, 0)$; $L = R$ points to an occurrence of c. If there is no such occurrence, we put $pos = posT = (root, 0)$; $L = R = -1$. The justification for such an update is Observation 12: since *maxlen* cannot be updated before the queue is emptied, we abandon the currently processed update of *pos* and skip some subsequent updates which cannot lead to the update of the answer (because $flag = 0$); then we use the tail (without the new symbol) to define the new value of $suff_r$.

Next we optimize the computations in the suffix tree using the dynamic weighted ancestor tree (DWAT) structure [17] for $\mathcal{T}$, which is maintained in parallel with $\mathcal{T}$; this tree has the size $\mathcal{O}(|\mathcal{T}|)$, and its update requires $\mathcal{O}(\frac{\log^2 \log |\mathcal{T}|}{\log \log \log |\mathcal{T}|})$ time per one update of the suffix tree.

DWAT was used to find implicit suffix links in the suffix tree for the order-preserving model [9] as follows: descend to the nearest node, follow its suffix link, and ascend using one $\mathcal{O}(\log\log|\mathcal{T}|)$-time query to DWAT. If we adopt this line, the total time spent by the algorithm will increase to $\Theta(n\log\log n)$. To avoid this, we exploit the following trick. First we try to find the implicit link by the usual procedure: ascend to a nearest node, follow suffix link, and descend the same distance. If after $\log\log|\mathcal{T}|$ operations we have not reached the destination edge, we abandon this try and find the link querying DWAT. This approach allows us to unite both bounds: $\mathcal{O}(\log\log n)$ update time from DWAT and $\mathcal{O}(|\mathcal{T}|) = \mathcal{O}(\frac{n}{E})$ total time for each conveyor, which gives $O(n)$ in total for all conveyors.

With all these optimizations, Algorithm 2 works in the same space as Algorithm 1; the update time is dominated by $\mathcal{O}(\log n)$ for the suffix tree update (line 8), including the update of DWAT; *lazyProcess* (line 4) requires only $\mathcal{O}(\log\log n)$ time per iteration, as shown above.

The total time of the tree update (including DWAT) is $\mathcal{O}(\frac{n}{E}\log n)$; summing this with the bound $\mathcal{O}(n)$ for all conveyors, we get the result of the theorem. ◀

▶ Remark 13. The heaviest operation in Algorithm 2 is the update of the suffix tree. However, in the literature we found no algorithm which gives better worst-case update time and is suitable for a polynomial integer alphabet. E.g., the algorithm by Breslauer and Italiano [4] has $\mathcal{O}(\log\log n)$ update time, but only for constant alphabets, while the algorithm of Kopelowitz [16] is based on the y-fast trie [22], which has $\mathcal{O}(\log n)$ worst-case update time.

5 Search of a Longest Repeat

Now we approach LRS. A simple solution is very close to Algorithm 1; in fact it is easier, because we do not need to take a special care about overlapping repeats: a symbol in the right occurrence of a repeat becomes available later than its counterpart from the left occurrence no matter whether the repeat is overlapping or not. It is enough to make the following changes in Algorithm 1: (i) use P_b instead of Q_b, (ii) build suffix tree left to right by Ukkonen's algorithm [20], enhanced with the dynamic perfect hashing, instead of Weiner's algorithm (line 9; line 8 is no longer needed), and (iii) redefine suff_r^i to be the longest suffix of P_r^i occurring in P_b^{i-r}. This gives us

▶ **Theorem 14.** *There is an algorithm solving, w.h.p.,* LRS *in a stream with a given additive error $E = \mathcal{O}(n^{0.99})$ in $\mathcal{O}(\frac{n}{E} + E)$ working space, $\mathcal{O}(n)$ total time, and $\mathcal{O}(\frac{n}{E})$ update time.*

However, there is a problem with enhancing this algorithm to get better update time. Namely, each iteration of Weiner's algorithm produces exactly one leaf and at most one internal node, while in Ukkonen's algorithm the number of new nodes at one iteration can be linear in the size of the tree. Hence for some updates of the tree $\mathcal{T}$ the time $\Theta(|\mathcal{T}|)$ can be inevitable (postponed updates may affect the search results). Instead, we work with Q_b and Q_r, comparing reversed strings of "reversed" hashletters. This allows us to stick to Weiner's construction and prove an analog of Theorem 11.

▶ **Theorem 15.** *There is an algorithm solving, w.h.p.,* LRS *in a stream with a given additive error $E = \mathcal{O}(n^{0.99})$ in $\mathcal{O}(\frac{n}{E} + E)$ space, $\mathcal{O}(n + \frac{n}{E}\log n)$ total time, and $\mathcal{O}(\log n)$ update time.*

We define pref_r^i be the longest prefix of Q_r^i occurring in Q_b^{i-r}, and redefine pos_r^i to be the position of pref_r^i in the current tree $\mathcal{T}(Q_b^i)$. Similar to Observation 7, we get

▶ **Observation 16.** *One has $\mathit{pref}_r^{i-1} = \mathit{pref}_r^{i-b}$ whenever $i + 1 \equiv_b r$.*

As with reversed repeats, after reading $S[i]$ we take $r = (i+1) \overline{\bmod}\, b$ and compute pos_r^{i-1} from $pos_r^{i-b} = (v, raise)$ walking up the tree until the edge with the position $raise$ symbols above v will be found. After this we compute pos_r^i, using a direct analog of Proposition 9.

▶ **Proposition 17.** *The longest proper suffix of $pref_r^i$ is a prefix of $pref_r^{i-1}$.*

Proof of Theorem 15. We define the block size $b = \lfloor \frac{E}{2} \rfloor$ and write $\mathcal{T}_j$ for the suffix tree $\mathcal{T}(Q_b^j)$. Algorithm 3 is presented below.

Algorithm 3 : FastAdditiveRepeat, ith iteration ($i \geq b$).

1: $r = i \bmod b + 1$
2: read $S[i]$; compute $I(i+1)$ from $I(i)$; $I = I(i+1)$
3: compute $FR = F^R(i-b+1, i)$ from I and $SI[r]$
4: update $pos[r] = pos_r^{i-b}$ to pos_r^{i-1} by walking up
5: compute $pos[r] = pos_r^i$ as longest prefix of $FR{\cdot}pref_r^{i-1}$ that exists in $\mathcal{T}$
6: update $answer$ by the value determined by $pos[r]$
7: **if** $r = b$ **then**
8: update $\mathcal{T}$ to $\mathcal{T}_i$, appending FR and building suffix links using BIS
9: $SI[r] = I$

Line 4 requires $\mathcal{O}(1)$ time, since only one internal node can appear in $\mathcal{T}$ between the corresponding iterations; line 8 requires $\mathcal{O}(\log n)$ time per iteration and thus $\mathcal{O}(\frac{n}{E} \log n)$ in total (see Algorithm 2). It remains to explain the computation in line 5. As in Theorem 11, we use two different ways to benefit from both good total time and good update time.

One way is to use *hard inverse links*, which are inverses of suffix links (i.e., link a node u to all nodes of the form au where a is a symbol). Clearly, they can be added to the tree simultaneously with suffix links and stored in a hash table similar to the one used for transitions by letters. To compute pos_r^i, one walks up the tree from the position pos_r^{i-1} until a node with a defined hard inverse link by the hashletter FR is reached, follows this link, and possibly walks some letters down inside one edge (however, the number of operations needed to compute the length of descent equals, in the worst case, the number of nodes passed on ascent). Note that one step up decreases the tree depth of a node by 1, the transition by hard link cannot increase it more than by 1, and the descent inside one edge does not affect it. Thus the total number of steps up in processing of a fixed trace is $O(|\mathcal{T}|)$. This gives us $\mathcal{O}(n)$ total time for all computations in line 5.

The second way is to use a one $\mathcal{O}(\log n)$-time query to BIS. Finally, the solution is to perform $\log n$ steps of the search in the tree, and if the requires position is not found, query BIS. This gives us both $\mathcal{O}(n)$ total time and $\mathcal{O}(\log n)$ update time. ◀

6 Lower Bounds

In this section we use Yao's minimax principle [23] to show the lower space bounds to the LRS problem, where the input length n and the input alphabet Σ are given; we use the notation $LRS_{|\Sigma|}[n]$. For LRRS, the same results can be obtained by straightforward adaptations of theorems from [11, Section 2], so we omit them. For LRS, the proofs of Lemma 20 and Theorem 22 also follow the scheme from [11], while Theorem 19 and Lemma 21 are proved by different arguments.

▶ **Theorem 18** (Yao's principle). *Let $\mathcal{X}$ be the set of inputs for a problem, $\mathcal{A}$ be the set of all deterministic algorithms solving it, and $c(a,x) \geq 0$ be the cost of running $a \in \mathcal{A}$ on $x \in \mathcal{X}$.*

Let p be a probability distribution over $\mathcal{A}$, and let A be an algorithm chosen at random according to p. Let q be a probability distribution over $\mathcal{X}$, and let X be an input chosen at random according to q. Then $\max_{x\in\mathcal{X}} \mathbf{E}[c(A,x)] \geq \min_{a\in\mathcal{A}} \mathbf{E}[c(a,X)]$.

First we prove that any Las Vegas online algorithm solving LRS with a given additive error needs at least linear space and thus cannot be used for streaming. For Las Vegas algorithms, the class $\mathcal{A}$ consists of all correct algorithms, and $c(a,x)$ is the memory usage.

▶ **Theorem 19.** *Let* A *be a Las Vegas streaming algorithm solving the problem* $\mathsf{LRS}_{|\Sigma|}[n]$ *with an additive error $E \leq 0.49n$ using $s(n)$ bits of memory. Then* $\mathbf{E}[s(n)] = \Omega(n \log |\Sigma|)$.

Proof. We give the proof for $\Sigma = \{0,1\}$ to simplify computations. It can be extended for arbitrary alphabet by encoding letters by binary strings of uniform length. Let $\mathcal{P}$ be the uniform distribution over all strings of even length n. Consider an arbitrary deterministic algorithm D, solving LRS with an additive error E. An input z is "good" if D spends at most $0.004n$ bits of memory on it, and "bad" otherwise. First assume that at least half of inputs are good. Then good inputs cover at least half, or $2^{0.5n-1}$, possible prefixes of length $n/2$ of inputs. After reading such a prefix of a good input, D is in one of at most $2^{0.004n}$ states. Then we can choose a class C, containing at least $2^{0.496n-1}$ prefixes sharing the same state of D. Note that there exist at most $n^2 \cdot 2^{0.495n}$ prefixes which contain a repeat of length $0.005n$. Further, any input string xx has a common substring of length $0.005n$ with at most $n^2 \cdot 2^{0.495n}$ strings of length $n/2$. Hence for n big enough, C contains two strings x and y such that neither of them contains a repeat of length $0.005n$ and xx has no common substring of length $0.005n$ with y. Then the length of the longest repeated substring in xx is $0.5n$, while for yx it is less than $2 \cdot 0.005n = 0.01n$. Since D is in the same state after reading x and y, it gives the same answer for xx and yx; one of the answers must be erroneous.

Thus our assumption about good inputs was wrong, and at least half of the inputs are bad. Then the expected memory usage of D is at least $0.004n/2 = \Omega(n \log |\Sigma|)$ bits. The reference to Theorem 18 finishes the proof. ◀

For Monte Carlo algorithms, $\mathcal{A}$ consists of all (non necessarily correct) algorithms, and $c(a,x)$ is the correctness indicator (0 if correct, 1 otherwise). First we show that the exact answer to LRS cannot be found using sublinear memory.

▶ **Lemma 20.** *There is a constant γ such that any Monte Carlo online algorithm solving* $\mathsf{LRS}_{\Sigma}[n]$ *exactly with probability $1 - \frac{1}{n}$ uses at least $\gamma n \log \min\{|\Sigma|, n\}$ bits of memory.*

The proof is similar to the proof of [11, Lemma 3] and can be found in Appendix.

For Monte Carlo algorithms with additive error, we first prove an auxiliary rough bound and then a sharp bound, using a reduction from the exact Monte Carlo algorithm.

▶ **Lemma 21.** *Any randomized Monte Carlo algorithm solving the problem* $\mathsf{LRS}_{|\Sigma|}[n]$ *with the additive error $E \leq 0.49n$ and the error probability $\frac{1}{n}$ uses $\Omega(\log n)$ bits of memory.*

Proof. Let $\Sigma = \{0,1\}$ and consider the uniform distribution $\mathcal{P}$ over any set P of $2^{n/2+1}$ strings of even length n with the following property: each string $x \in \Sigma^{n/2}$ occurs twice as the left half of a string from P such that one of these strings is xx and the other is xy where y is a fully randomly chosen string of length $n/2$. Thus, half of strings in P have repeats of length $n/2$, and the other strings are fully random and thus their longest repeats are of length $\Theta(\log n)$ with probability $1 - \frac{1}{n}$ (cf. [11, Lemma 22]). Hence if an algorithm stores

any sketch of $s(n) = o(\log n)$ bits of information about x, it distinguishes between x and y in the second part with a too small probability: y shares this sketch with x with probability $2^{-s(n)} > \frac{1}{n}$. ◀

▶ **Theorem 22** (Monte Carlo additive approximation). *Any randomized streaming Monte Carlo algorithm solving the problem* $\mathsf{LRS}_{|\Sigma|}[n]$ *with the additive error* $E \leq 0.49n$ *with probability* $1 - \frac{1}{n}$ *uses* $\Omega(\frac{n}{E} \log \min\{|\Sigma|, \frac{n}{E}\})$ *bits of memory.*

Proof. Let $\sigma = \min\{|\Sigma|, \frac{n}{E}\}$. Since the bound $\Omega(\frac{n}{E} \log \sigma)$ becomes $\Omega(\log n)$ if $E = \Omega(\frac{n \log \sigma}{\log n})$, we further suppose that $E = o(\frac{n \log \sigma}{\log n})$.

Assume that there is a Monte Carlo streaming algorithm A solving $\mathsf{LRS}_{|\Sigma|}[n]$ with additive error E with probability $1 - \frac{1}{n}$, using $o(\frac{n}{E} \log \sigma)$ bits of memory. Let $n' = \lfloor \frac{n-E}{E+1} \rfloor$. We define new Monte Carlo streaming Algorithm A', which processes a string $x[1..n']$ as follows: run Algorithm A on $x' = 0^E x[1] 0^E x[2] \cdots 0^E x[n'] 0^E$, using $\log E \leq \log n$ additional bits of memory to count to E, get an answer R, and return the number $\lfloor \frac{R}{E+1} \rfloor$. If the longest repeat in x has length r, the longest repeat in x' has length $(r+1)E + r$ (each occurrence consists of r letters of x and $r+1$ blocks of 0's). Since Algorithm A has additive error E, one gets $r(E+1) \leq R \leq (r+1)E + r$, so A' must return r. Hence Algorithm A' solves $\mathsf{LRS}_{|\Sigma|}[n']$ exactly with probability $1 - \frac{1}{n} \geq 1 - \frac{1}{n'}$ using $o(n' \log \sigma) + \log n$ bits of memory. By the above assumption on E, this number is $o(n' \log \sigma)$, contradicting Lemma 20, which requires the memory usage for exact LRS to be $\Omega(n' \log \sigma)$. ◀

7 Conclusion

In this paper, two classical string problems (LRS and LRRS) are considered in the streaming model. We proved that they can be solved only by Monte Carlo approximation algorithms; for the additive approximation error, we presented efficient algorithms which are space optimal whenever $E = \mathcal{O}(\sqrt{n})$ and $|\Sigma| = \Omega(n^{0.01})$. The algorithms are based on suffix trees, which is rather exotic for the streaming model.

Two intriguing open problems about LRS and LRRS streaming solutions are the existence of algorithms with the additive error $E > \sqrt{n}$ within $o(\sqrt{n})$ memory and the existence of efficient algorithms with multiplicative error. We also note that an efficient solution of the LRRS problem can be important for solving the problem of the longest gapped palindrome.

References

1 Amihood Amir, Tsvi Kopelowitz, Moshe Lewenstein, and Noa Lewenstein. Towards real-time suffix tree construction. In *International Symposium on String Processing and Information Retrieval*, pages 67–78. Springer, 2005.

2 Alberto Apostolico, Costas Iliopoulos, Gad M. Landau, Baruch Schieber, and Uzi Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3(1-4):347–365, 1988.

3 Dany Breslauer and Zvi Galil. Real-time streaming string-matching. In *Combinatorial Pattern Matching*, volume 6661 of *LNCS*, pages 162–172, Berlin, 2011. Springer.

4 Dany Breslauer and Giuseppe F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. *Journal of Discrete Algorithms*, 18:32–48, 2013.

5 Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya. Dictionary matching in a stream. In *Algorithms-ESA 2015*, pages 361–372. Springer, 2015.

6 Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya. The k-mismatch problem revisited. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 2039–2052. Society for Industrial and Applied Mathematics, 2016.

7 Raphaël Clifford, Tomasz Kociumaka, and Ely Porat. The streaming k-mismatch problem. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1106–1125. SIAM, 2019.

8 Raphaël Clifford and Tatiana Starikovskaya. Approximate Hamming distance in a stream. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016*, volume 55 of *LIPIcs*, pages 20:1–20:14, 2016.

9 Maxime Crochemore, Costas S Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Order-preserving indexing. *Theoretical Computer Science*, 638:122–135, 2016.

10 Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. Dynamic hashing in real time. In *Informatik*, pages 95–119. Springer, 1992.

11 Paweł Gawrychowski, Oleg Merkurev, Arseny M. Shur, and Przemysław Uznański. Tight Tradeoffs for Real-Time Approximation of Longest Palindromes in Streams. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016*, volume 54 of *LIPIcs*, pages 18:1–18:13, 2016.

12 Shay Golan, Tsvi Kopelowitz, and Ely Porat. Towards optimal approximate streaming pattern matching by matching multiple patterns in multiple streams. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

13 Shay Golan and Ely Porat. Real-Time Streaming Multi-Pattern Search for Constant Alphabet. In *25th Annual European Symposium on Algorithms, ESA 2017*, volume 87 of *LIPIcs*, pages 41:1–41:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

14 Elena Grigorescu, Erfan Sadeqi Azer, and Samson Zhou. Streaming for aibohphobes: Longest palindrome with mismatches. In *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017*, volume 93 of *LIPIcs*, pages 31:1–31:13, 2017.

15 Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM journal of research and development*, 31(2):249–260, 1987.

16 Tsvi Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *Foundations of Computer Science (FOCS), 2012 IEEE 53rd Annual Symposium on*, pages 283–292. IEEE, 2012.

17 Tsvi Kopelowitz and Moshe Lewenstein. Dynamic weighted ancestors. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 565–574. Society for Industrial and Applied Mathematics, 2007.

18 Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *Foundations of Computer Science, 2009. FOCS'09. 50th Annual IEEE Symposium on*, pages 315–323. IEEE, 2009.

19 Jakub Radoszewski and Tatiana A. Starikovskaya. Streaming K-Mismatch with Error Correcting and Applications. In *2017 Data Compression Conference, DCC 2017*, pages 290–299. IEEE, 2017.

20 E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

21 Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.

22 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17:81–84, 1983.

23 Andrew Yao. Probabilistic computations: Toward a unified measure of complexity. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 222–227, 1977.

A Appendix

Proof of Lemma 20. We use the auxiliary problem $\mathsf{LCP}_\Sigma[n]$ which is to find the longest common prefix of left and right halves of the input. First we prove that if A is a Monte Carlo online algorithm solving $\mathsf{LCP}_\Sigma[n]$ exactly using less than $\lfloor \frac{n}{2} \log |\Sigma| \rfloor$ bits of memory, then its error probability is at least $\frac{1}{n|\Sigma|}$.

By Theorem 18, it is enough to construct probability distribution $\mathcal{P}$ over Σ^n such that for any deterministic algorithm D using less than $\lfloor \frac{n}{2} \log |\Sigma| \rfloor$ bits of memory, the expected probability of error on a string chosen according to $\mathcal{P}$ is $\geq \frac{1}{n|\Sigma|}$. To do this, let $n' = \frac{n}{2}$. For any $x \in \Sigma^{n'}$, $k = 1, \ldots, n'$, $c \in \Sigma$, let $w(x,k,c) = x[1..n']x[1..k{-}1]cx[k{+}1..n']$. Now $\mathcal{P}$ is the uniform distribution over all such $w(x,k,c)$. Choose an arbitrary maximal matching of strings from $\Sigma^{n'}$ into pairs (x,x') such that D is in the same state after reading either x or x'. At most one string per state of D is left unpaired, that is at most $2^{\lfloor \frac{n}{2} \log |\Sigma| \rfloor - 1}$ strings in total. Since there are $|\Sigma|^{n'} = 2^{n' \log |\Sigma|} \geq 2 \cdot 2^{\lfloor \frac{n}{2} \log |\Sigma| \rfloor - 1}$ possible strings of length n', at least half of the strings are paired. Let s be the longest common prefix of x and x', so $x = scv$, $x' = sc'v'$, where $c \neq c'$ are distinct letters. Then D returns the same answer on $w(x,|s|,c)$ and $w(x',|s|,c)$, although $\mathsf{LCP} = |s|$ in one case and $\mathsf{LCP} \geq |s|+1$ in the other. Similarly, D errs on either $w(x,|s|,c')$ or $w(x',|s|,c')$. Thus the error probability is at least $\frac{1}{2n'|\Sigma|} = \frac{1}{n|\Sigma|}$.

Now we prove the lemma for $LCP_\Sigma[n]$ with an amplification trick. Assume we have a Monte Carlo streaming algorithm, which solves $LCP_\Sigma[n]$ exactly with error probability ε using $s(n)$ bits of memory. Then we can run its k instances simultaneously and return the most frequent answer. The new algorithm uses $\mathcal{O}(k \cdot s(n))$ bits of memory and its error probability ε_k satisfies the inequality $\varepsilon_k \leq \sum_{2i<k} \binom{k}{i} (1-\varepsilon)^i \varepsilon^{k-i} \leq 2^k \cdot \varepsilon^{k/2} = (4\varepsilon)^{k/2}$. Let $\kappa = \frac{1}{6} \frac{\log(4/n)}{\log(1/(n|\Sigma|))}$, so $\kappa = \frac{1}{6} \frac{1-o(1)}{1+\log|\Sigma|/\log n} = \Theta\left(\frac{\log n}{\log n + \log|\Sigma|}\right) = \gamma \cdot \frac{1}{\log|\Sigma|} \log\min\{|\Sigma|, n\}$ for some constant γ. Assume that A uses less than $\kappa \cdot n \log|\Sigma| = \gamma \cdot n \log\min\{|\Sigma|, n\}$ bits of memory. Then running $\lfloor \frac{1}{2\kappa} \rfloor \geq \frac{3}{4} \frac{1}{2\kappa}$ (which holds since $\kappa < \frac{1}{6}$) instances of A in parallel requires less than $\lfloor \frac{n}{2} \log|\Sigma| \rfloor$ bits of memory. But then the error probability of the new algorithm is bounded from above by $\left(\frac{4}{n}\right)^{3/16\kappa} = \left(\frac{1}{n|\Sigma|}\right)^{18/16} \leq \frac{1}{n|\Sigma|}$, which we have already shown to be impossible.

The lower bound for LCP can be converted into a lower bound for solving LRS exactly by padding the input so that the longest repeat is the common prefix of the whole string and its right half. Let $x = x[1..n]$ be the input for $\mathsf{LCP}_{|\Sigma|}[n]$, with the answer k. We define $w(x) = 0^n 1 x[1..\frac{n}{2}] 0^n 1 x[\frac{n}{2}+1..n]$, where $0, 1 \notin \Sigma$; clearly, $w(x)$ contains a palindrome of length at least $n+k+2$. On the other hand, any repeated substring of $w(x)$ of length $\geq n{+}1$ must contain the substring $0^{n/2}1$, which has just two occurrences in w. Thus we have reduced solving $\mathsf{LCP}_{|\Sigma|}[n]$ to solving $\mathsf{LRS}_{|\Sigma|}[3n+2]$. We already know that solving $\mathsf{LCP}[n]$ with probability $1 - \frac{1}{n}$ requires $\gamma \cdot n \log\min\{|\Sigma|, n\}$ bits of memory, so solving $LRS_{|\Sigma|}[3n+2]$ with probability $1 - \frac{1}{3n+2} \geq 1 - \frac{1}{n}$ requires $\gamma \cdot n \log\{|\Sigma|, n\} \geq \gamma' \cdot (3n+2) \log\min\{|\Sigma|, 3n+2\}$ bits of memory. The reduction needs $\mathcal{O}(\log n)$ additional bits of memory to count up to n, but for large n this is much smaller than the lower bound if we choose $\gamma' < \frac{\gamma}{4}$. ◀